Interaction of objects in a virtual environment: a two-point paradigm

Steve Bryson

RNR Technical Report RNR-91-009, February 1991

Computer Sciences Corporation/
Applied Research Office, Numerical Aerodynamics Simulation Division
NASA Ames Research Center
MS T045-1
Moffett Field, Ca. 94035
bryson@nas.nasa.gov

Abstract

Virtual environments are computer generated environments with computer generated objects. These objects will, in general, have interactions. Thus the construction and editing of virtual environments will involve the specification of interactions. This paper presents a way of viewing interactions, the two-point paradigm, which permits the user to view and edit interactions between pairs of objects in real time from within the environment. The two-point paradigm naturally suggests data structures for objects, interaction definition, and the actual interactions in the environment. These data structures are described, and a simple editor based on them is suggested.

1: INTRODUCTION

1.1: Purpose and motivation

Virtual environments are a new approach to human-computer interfaces, involving the integration of various input and output devices to provide the user with the illusion of being immersed in a computer generated environment. Typical virtual environment systems[1, 2, 3, 4] use inputs that track the user's head and hand positions and orientations and output displays that fill the user's field of view. The rendering of the virtual environment is from a point of view controlled by of the user's head position and orientation. The tracking of the user's hand allows the manipulation of virtual objects in a way that creates the illusion of holding and manipulating a real object.

As virtual environments create a compelling illusion of a real, interactive world, the question of defining and characterizing the interactions between objects in the environment is of great interest. In particular, there is a desire to allow the user to assign these interactions from within the environment in real time. This would allow the user to construct environments from the inside. The structure defined in this paper, the two point interaction paradigm, is designed to allow a user to arbitrarily add objects to a virtual environments and edit their interactions.

Abstractly, a virtual environment is a collection of computer generated objects which interact with themselves and the user. Specifying a virtual environment involves specifying both the appearance of the objects in that environment and their behavior. As the objects are computer generated, the behavior of objects is defined by some code which updates the graphic representation of the object. Usually that code will act on some data associated with the objects. The problem of specifying the interactions between objects is the problem of specifying the functions defining those interactions, the objects on which they act, and the object data that is affected.

The problem of editing the interactions between objects in virtual environments is, in general, more difficult the greater the number of objects. The possible complexity of data paths grows very rapidly with the number of objects, which makes the display of the current state of interactions in an environment difficult. Further, the impact of modifications to the interactions can be difficult to anticipate.

The two point interaction paradigm addresses these problems be considering interactions exclusively between pairs of objects. Thus to consider the interactions that an object is participating in, it is sufficient to display all the objects with which that object interacts. Once a pair of objects is specified, the interactions between those two objects can be listed in the order in which they are executed.

The purpose of this paper is to describe the underlying data structures and the run-time evaluation of those structures. A possible external interface for the editing of these structures will be described. Generalizations of this structure will be briefly discussed.

1.2 Brief survey of existing approaches to the interaction problem

The problem addressed by the two point paradigm is that of keeping track of the evolution of the data defining a system as it is acted on by various pieces of code. There have been several other approaches to this problem.

One approach that has been implemented by VPL Research, Inc. in their commercial product is the data flow paradigm. Blanchard et. al.[1] have implemented a visual interface to a data flow system that allows for real-time manipulation of the flow of data in a virtual environment. In this case interactions are treated as paths for the movement and manipulations of data. The display and manipulation of the state of the data flow is, however, outside the virtual environment.

Another approach is the idea of object oriented programming. This approach is being used in the structure of some programming languages. The author is not aware of any implementations of real-time interaction editors for virtual environments based on the object oriented paradigm, however. The two-point paradigm described here is very close in philosophy to the object oriented approach, and would be most naturally implemented in an object oriented language.

1.3 Brief description of the two-point paradigm

The interaction paradigm introduced in this paper is inspired by the classical description of interaction in real world physics. In the real world, one can take the interaction of objects as due to forces that act pair-wise between physical objects. While many forces may act on an object simultaneously, the net action of these forces can be taken to be the linear sum of the forces on that object from each other object. Thus one can obtain a complete description of the net force on an object in terms of the individual forces on that object due to each other individual object. These individual forces are always between pairs of objects.

In the two-point paradigm, the pair-wise interaction idea is generalized beyond forces to all interactions. Thus an interaction becomes a function that takes as input the state of the two objects interacting and modifies the state of one of them. To determine the state of an object after all interactions, one computes all the interactions for all other objects in the virtual environment.

The state of an object is implemented as a data structure containing a set of parameters. The parameters in that object's data structure will depend on the kind of that object. For example, a particle object would have a position, velocity, mass, graphic representation and so on. A sound object would have a position, velocity, volume, sound representation, etc. A field object would have a bounding volume, field data as a function of position, and a graphic representation.

An interaction would have a pointer to a piece of code, and data that would indicate what types of object data that interaction can act on. Note that interactions are defined independently of the actual objects that these interactions will act on. Interactions are defined in terms of kinds of objects.

The description of the interactions in the virtual environment is a list of interactions for every pair of objects. This is most naturally thought of as a matrix of interactions, with the rows and columns of the matrix corresponding to the objects in the virtual environment, and the entries of the matrix being a list of interactions between the objects for that row and column (fig. 1). This matrix is called the interaction matrix. The evaluation of the interactions in the environment is the execution of each entry of the interaction matrix, passing to the interaction routines the row and column objects.

	Object 1 (floor)	**Object 2** (ball)
Object 1 (floor)	null	Gravity bounce
Object 2 (ball)	null	Integrate Equations of Motion

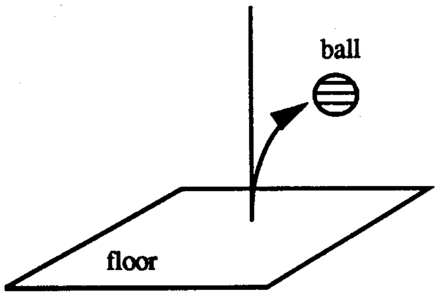

Fig 1: An example interaction matrix, with a ball and a floor being the two objects. The ball is acted on by the floor in two ways: gravity pulling it down and bouncing which reverses the z component of velocity. The floor is not acted on by the ball. The ball's diagonal entry updates its velocity from its acceleration and its position from its velocity.

The two-point paradigm has several ambiguities, mostly concerning the ordering of the interactions. For any type of interaction, as the matrix is evaluated and the object states are updated, the order in which the interactions are computed can dramatically effect the outcome of the sum of interactions. For example, if an interaction moves one object and another interaction causes that object to move another object, then the final position of the latter object would depend on the order in which these interactions were performed. A double buffering of the object data, defining read and write fields of all object data (which switch after each evaluation of the interaction matrix) solves this problem. This will make the environment act as if all interactions occurred simultaneously.

A deeper ordering problem concerns forces which are not linearly incremental in their action. Linearly incremental means that the interaction adds a quantity to some field of the object data. For linearly incremental interactions, the order of evaluation is not important as addition is commutative. There may, however, be other interactions that can multiply, or even replace the values of fields in the object data. These interactions will be extremely order dependant. Thus when non-linear or non-incremental interactions are used great care must be taken by the user so that the desired result occurs.

2: THE INTERACTION MATRIX

2.1 Definitions

An **object** is a data structure with several fields. An object can be one of several kinds, with each kind having different fields in its object structure (fig. 2). The first field of the object data would be the object type identifier. Other fields would include data defining the state of the object. This data would have meaning only in the context of the interactions on this object and the object's data representation, but for the purposes of clarity in this paper several generic fields will be referred to. For example, a particle object type may contain position, velocity, acceleration, mass, and color. A field object type may contain position, velocity, and a function of position that would return the value of the field at that position.

Ball:	Vector Field:	User's Hand:
-- position vector	-- bounding volume	-- position vector
-- velocity vector	-- field parameters	-- finger joint array
-- mass	-- field outputs	-- current gesture
...	-- scale	...
	...	

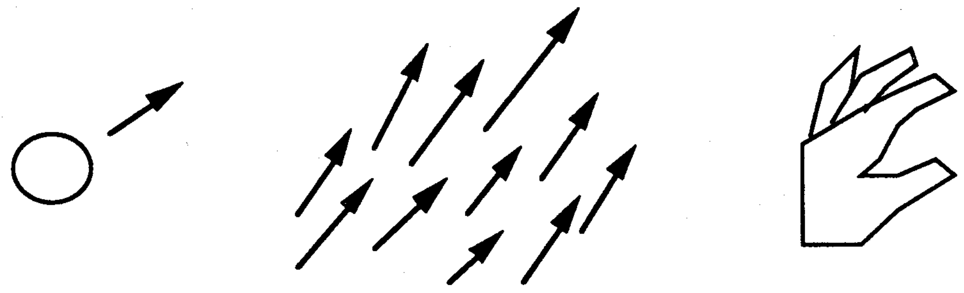

Fig 2: Examples of different kinds of objects with different data structures.

An **interaction** is a data structure that includes several fields. The first is a function, the interaction itself, which takes as input an ordered pair of object data and (by convention) modifies the data of the second object of that pair. The following notation will be adopted for an interaction:

$$\text{interaction} : (O1, O2) \rightarrow O2$$

Here, O1 and O2 refer to the object data structures, and the arrow indicates that the values of the data fields are modified by the interaction. Thus an example of interaction code would be, using C syntax,

```
example(object *O1, object *O2)
{
        O2->position = O1->position + O2->velocity;
}
```

In a simple implementation of this paradigm, each interaction would be defined for only certain types of objects, and so the next field of the interaction data structure would be the object types for which this interaction is defined. This be an ordered pair of lists of types, as an interaction may be defined for more then one type of object. In the above example, the interaction would be defined for any pair of objects such that the first object has a position, and the second object has both a position and a velocity. In this simple implementation, the interaction would be passed pointers to the object data structures, and the reference to the fields in those data structures would be hard coded.

An **interaction matrix entry** is a list of interactions which are evaluated in sequence. When the interactions act on the objects directly, this is all that is required of the matrix entry. The ordered pair of objects acted on by the interaction is defined to be the row and column objects for the current interaction matrix entry. In this way there is a distinction between, for example, object a acting on object b and object b acting on object a. These actions need not have anything to do with each other, or in other words the interaction matrix need not be symmetric.

An interesting case is the diagonal entries of the interaction matrix, where the object would be interacting with itself. By convention, these diagonal entries are taken to be those functions which are required to update the object. Examples of such functions include integrating the velocity to update position, performing any graphic operations required by this object, etc. Due to the special nature of the diagonal entries, ordering problems have special prominence. The user would typically want the operations updating the object to take precedence over any other interactions, and so by convention the diagonal entries are evaluated after all the off diagonal entries.

2.2 Run-time structure

During the running of a program designed around the two-point paradigm, the interactions between the objects is computed by running through the list of interactions for each entry of the interaction matrix. Each evaluation of the interaction matrix is called a frame. In order to alleviate the ordering problem, there are two copies of all object data structures, which alternate between the read data and the write data. In each frame the interaction routines read the data from the read half of the object data, and modify the data in the write half of the object data. This is implemented by passing the pointer to a two entry array of object data, with the choice of read or write entry conrtolled by a global variable called buffer. Buffer takes on the values 0 or 1. The example described in section 2.1 above becomes:

```
example(object *O1[2], object *O2[2])
{
        O2[(buffer+1)mod2]->position = O1[buffer]->position
                + O2[buffer]->velocity;
}
```

After each frame, the value of buffer is toggled.

2.3 Generalization to more versatile interactions

The structure as defined thus far allows the assignment of interactions to objects, where the interaction is predefined to act on certain fields of the objects' data structure. If the user wished to assign the action of an interaction to a different field of the object data structure, an entirely new interaction function would need to be defined. A generalization that would allow the user to determine which fields of the objects' data structure as part of the interaction's data is described in this section. This would allow the user to experiment in very general ways with the interactions in a virtual environment. In this way non-standard virtual physics can be explored, and the effects of interactions can be mapped to the most convenient object property. Examples include velocity dependant gravity and coloring an object according to the strength of a field.

In the structure described in section 2.1, an interaction is defined as a data structure which contains a function call to the interaction itself. This function took as arguments the two objects which this interaction was acting upon. These objects are identified with the row and column of the interaction matrix entry which contains this interaction. Which fields of the objects' data structures was acted upon was determined by the interaction function itself. In the generalization suggested here, an interaction does not take the objects as arguments, but rather takes the fields of the objects.

The interaction would consist of a data structure with fields that include: The function call itself, which would expect two arguments each providing a list of values that would be acted on by the function, the first list being the read values and the second list being the write values; the number of values that this function requires; and a list of the types (i.e. integer for the first argument, vector for the second argument, etc.) of arguments that this function requires.

The generalized interaction matrix entry would contain: an array of two lists of pointers to the actual fields of the objects' data structure; and a pointer to the interaction structure defined in the above paragraph. As above, the objects involved would be determined by the row and column of the current entry. The arrays are required so that the read and write fields can be accessed separately. These lists would be constructed when the user specifies which objects were participating in the interaction, which interaction is taking place, and which fields in the object data structures are being passed to the interaction, and which field(s) are to be modified by the output of the function.

The run-time execution of the interaction matrix would, in this more general case, involve calling the function in the interaction structure pointed to by each matrix entry with a pointer to the current read parameters as the first argument and a pointer to the current list of write values as the second argument.

While this generalization greatly enhances the ability of the user to manipulate interactions, it requires more user input to specify the interaction. Thus some kind of default structure should be worked out, which would allow the user to define the interactions much like the less general system described in section 2.1 above when desired.

3: INSERTING NEW OBJECTS INTO THE INTERACTION MATRIX

Inserting new objects into the virtual environment has the effect of adding a new row and column to the interaction matrix. Each entry of the new row and column of the interaction matrix must then be specified for the new object to interact with the other objects in the environment. When there are many objects in the environment, however, this task could be very time consuming. The problem of assigning default values for the entries of the interaction matrix has motivated the concept of object type.

An object type is defined to be two lists of interactions for every type of object. Two lists are required because interactions are ordered pairs, so that the interaction between objects a and b may not be the same as the interaction between objects b and a. Thus object types are defined in terms of each other via interactions. An example of an object type list is given for the type ball. This type would, for example, include the following interactions for the following other types of objects:

other object type	interaction when 1st object	interaction when 2nd object
wall	null	bounce
floor	null	fall and bounce
ball	gravitational force	gravitational force
self	integrate acceleration and velocity	

When an object of a certain type is inserted into the virtual environment, the entries of the interaction matrix in the new row and column for that object are determined by the following algorithm. For each (column) object in the new row corresponding to this object, the type of the column object is used as an index into the new object's table of default interactions. The same procedure is used to set the interaction matrix entries in the new object's row.

There is another, more formal way, of thinking ob object type as an assignment of a list of interactions to an ordered pair of predefined object types. The the new interaction matrix entries would be filled by the lists corresponding to the (row, column) object pairs for that entry.

In the generalization described in section 2.3, the types would be defined with a list of the fields for each interaction in addition to the list of the interactions themselves.

4. EDITING STRUCTURES

The point of the software structure described in this paper is the ability of the user to modify the interactions between objects in the virtual environment in real time. This requires an editing interface to be constructed on top of the structures described above.

An example of such an editing interface would be primarily driven by 3-dimensional menus, with objects indicated by pointing with some 3 dimensional tracking device. The menus would include: a menu listing the available types of objects, perhaps with choices of graphics representation; a menu listing the current interactions once a pair of objects is specified; and a menu listing available interactions.

A typical editing session would go as follows (fig. 3). Assume that the virtual environment already has some objects defined with interactions. Let's say that the user wishes to introduce some new objects into the environment and edit the interactions. The object selection menu would be accessed and an object chosen. This object would appear in the environment with some default graphic representation. At this time the object would be added to the interaction matrix, with the new entries of the matrix being set to the defaults as determined by the type of the object. Now say that the user wishes to add new a interaction to the interactions between two objects. The user would first point to one object, then the second object. The order of pointing would determine which object is the second object in the ordered pair sent to the interaction and so is the object acted upon. If the user wishes to review the existing interactions between these objects, a menu is presented which shows the list of interactions in the interaction matrix entry for these two objects. Selecting one of these items indicates where in the interaction list the user's actions should occur. If the user wishes to delete an interaction, the indicated interaction is deleted. If the user wishes to add an interaction, the menu of available interactions is accessed, and the user selects which interaction to add. This interaction is inserted into the interaction list in the interaction matrix entry at the entry selected. Similarly, the interactions in the interaction list may be replaced, permuted, copied and so on.

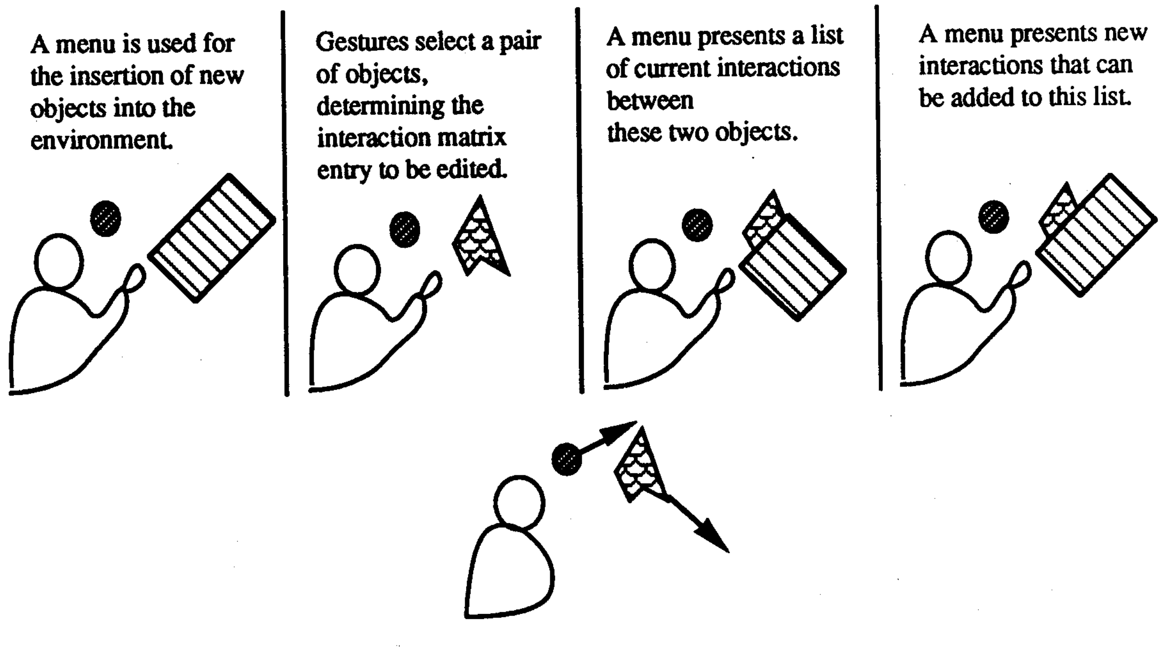
A menu is used for the insertion of new objects into the environment.
Gestures select a pair of objects, determining the interaction matrix entry to be edited.
A menu presents a list of current interactions between these two objects.
A menu presents new interactions that can be added to this list.

Fig 3: Various editing steps in an example implementation of an interaction editor based on the two-point paradigm.

The above scenario requires some additional structure. The interaction data structures should have text descriptions which identify the interaction, as well as a unique identifier. Then each interaction matrix entry will contain a list of identifier identifiers corresponding to the interaction list for this entry. In this way the current list of interactions can be constructed as a text menu.

It may be desired that the user have the capability of manipulating parameters in the interactions themselves. If a parameter in an interaction is to be manipulated globally, such the strength of the gravitational force, it is sufficient that the interaction have its own interface which would allow the manipulation of its parameters. When the variation in the parameters of the interaction are to be local, those parameters become parameters of the object involved rather than the interaction. This dovetails into the qeneralization described in section 2.3, where the data fields of the objects are themselves passed to the interaction function. This motivates us to design interactions so that parameters of the interactions such as strength, exponents, and so on are inputs to the interaction function.

5. CONCLUSIONS

The two-point paradigm described here is a simple way of organizing the display and manipulation of interactions between objects in a virtual environment. From the demand that editing be simple and intuitive, several data structures are motivated. These structures in turn naturally suggest the implementation of editors for interactions in virtual environments. This in combination with a powerful 3D graphics editor would lead to a very powerful tool for the construction and continual modification of virtual environments. It is hoped that such a system would promote great creativity in the construction of interesting, educational, and entertaining virtual environments.

There is no claim, however, that the two-point paradigm is a complete answer to the interaction problem. There are presumably operations on virtual environments that are clumsy to describe as interactions between pairs of objects. For this reason a full virtual reality development system would also have other ways of approaching the evolution of the virtual environment.

The two-point paradigm and its data structures do suggest a level of abstraction where the interactions between objects can be standardized. In this way one can define standard interactions much as one currently defines standard graphics primitives.

Another place where the two-point paradigm may be relevant is the development of virtual reality networks. One possibility is to send the interactions and object types across the network. The receiving workstation can compute the evolution of the environment, obviating the need to be continually sending the changing positions of objects across the network. As only changes in the interactions and addition of objects need be sent over the network, the bandwidth demands are greatly decreased.

By bringing the interactions between objects in a virtual environment to the fore and giving the end user control over these interactions, it is hoped that users will become participants in the creative process rather than spectators of other's work. A notable and successful example of such a system is Warren Robinett's Rocky's Boots[5], which allows the user to construct robots which interact with their virtual environment. Virtual environments hold the promise of making such systems part of our everyday lives.

6. ACKNOWLEDGEMENTS

The author would like to acknowledge Ian MacDowall for the conversations on the path which lead to the two-point paradigm. Thanks also to Rick Jacoby and Steve Ellis of the Aerospace Human Factors Research Division at NASA Ames for useful discussions and assistance, particularly for the use of the VIEW facility where the example program was developed. The author would also like to thank his colleagues in the Advanced Research Office of the NAS division at NASA Ames for their many helpful comments and discussions, particularly Sam Uselton, Creon Levit, Tom Lasinski, Al Globus, Jeff Hultquist, and Horst Simon. Finally, particular thanks go to Scott Fisher, without whom none of this would have happened.

7. REFERENCES

1 Blanchard, C. et. al., Reality Built for Two: A Virtual Reality Tool, *1990 Symposium on Interactive 3D Graphics*, Computer Graphics, Vol. 24, No. 2, March 1990

2 Bryson, S. and Levit, C., Virtual Wind Tunnel, *RNR Technical Report #*, Vol 24, #4, July 1990, pp 187-193

3 Fisher, S. et. al., Virtual Environment Interface Workstations, *Proceedings of the Human Factors Society 32nd Annual Meeting*, Anaheim, Ca. 1988

4 Fisher, S., Virtual Environments, Personal Simulation and Telepresence, *Implementing and Interacting with Real Time Microworlds*, Course Notes, Vol. 29, SIGGRAPH 1989

5 Robinett, W., *Rocky's Boots* (educational software product) The Learning Company, Fremont, CA. 1982